Calculating the Impedance of Alternating-Current Circuits in C++

Mohan Deo

May 2022

Abstract

Code was written in C++ 17 making use of polymorphism and inheritance to calculate the impedance of AC circuits. This involved recursive functions to allow for nested circuits to be created. Memory management was handled by the use of shared pointers. Resistors, capacitors and inductors (as well as their non-ideal counterparts) could be added to the AC circuit. Circuit diagrams can also be printed to the terminal. The impedance results from the program show strong agreement with calculations using other methods.

1. Theory

A circuit driven by a current which continually reverses direction is referred to as an alternating current (AC) circuit. This results from the application of an alternating potential applied at two terminals of components in the circuit. The use of AC allows for the implementation of a device known as a transformer. This makes the distribution of electrical power much more economical than for direct current [1]. Due to this, AC is used across the national power grid in the UK. Therefore, anything which plugs directly into the grid also has AC. This means AC is applicable to many devices.

An important relation which holds for AC circuits is Ohm's law,

$$V = IZ \quad , \tag{1}$$

where V and I are the voltage and current applied respectively and Z is the impedance. Since the voltage changes direction with a given frequency, it can be represented in the complex plane. The same argument can be applied for the current. Therefore, the impedance is also a function of the driving frequency and is represented as a complex number. Physically, the impedance relates to the amount of opposition to AC flow [2].

The impedance has different expressions for each type of component (Table 1). Since it is a complex number, it possesses an associated with a phase angle. This is referred to as a phase-shift and is the difference between the voltage and current phases.

Element	Impedance	Phase shift
Ideal resistor	R	0
Ideal capacitor	$-i\frac{1}{\omega C}$	$\frac{\pi}{2}$
Ideal inductor	$iL\omega$	$-\frac{\pi}{2}$

Table 1 - A summary of all the impedances and phase shifts ideal resistors, capacitors and inductors. The quantities R, C and L represent resistance, capacitance and inductance respectively. The driving frequency of the circuit is given by ω [2].

None of the expressions in Table 1 account for effects which take place in real (non-ideal) components. For instance, wire-wound resistors have an inductance associated with them [3]. This is often called ‘parasitic inductance’. To account for this, the resistor is constructed from an ideal resistor connected to an inductor. The equivalent circuits for non-ideal resistors, capacitors and inductors are shown in Figures 1, 2 and 3.

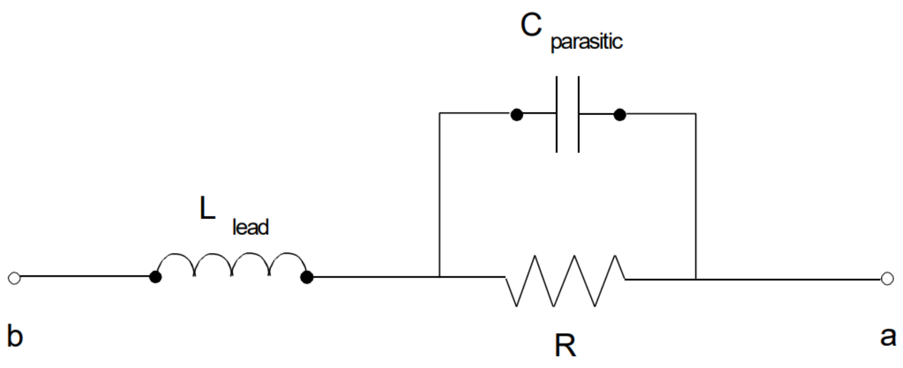


Figure 1 – Equivalent circuit for a non-ideal resistor [3].

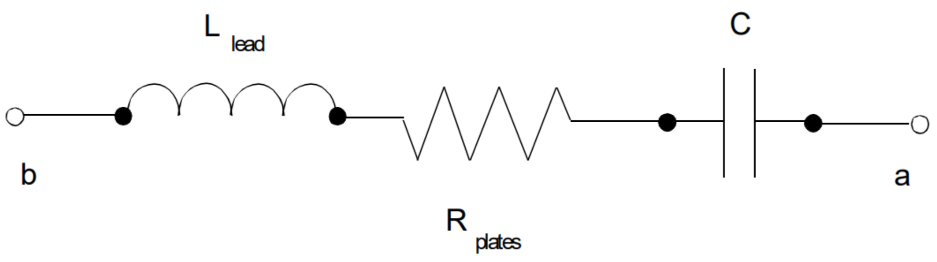


Figure 2 – Equivalent circuit for a non-ideal capacitor [3].

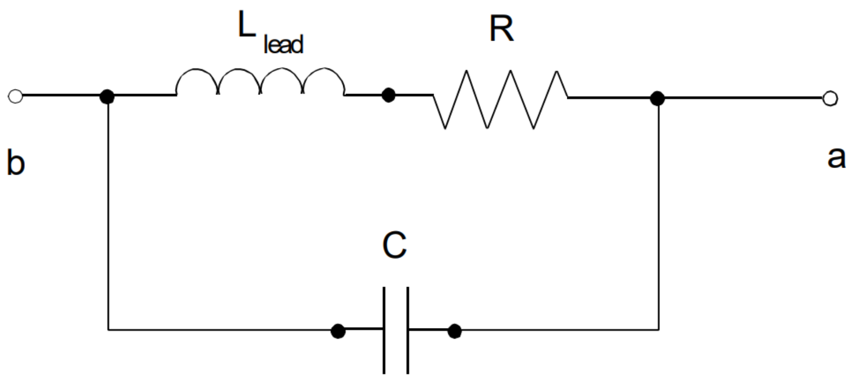


Figure 3 - Equivalent circuit for a non-ideal inductor.

Each of these circuits now allows for expressions for impedance of non-ideal components to be created (Table 2) [3].

Element	**Impedance**
Non-ideal resistor	$\frac{i\omega L_{lead} + R(1 - \omega^2 L_{lead} C_{par})}{i\omega R C_{par} + 1}$
Non-ideal capacitor	$R_{plates} + i(\omega L_{lead} - \frac{1}{\omega C})$
Non-ideal inductor	$\frac{i\omega L + R_{par}}{1 - \omega^2 L C_{par} + i\omega C_{par} R_{par}}$

Table 2 – A summary of the impedances of non-ideal components [3].

2. Software design

To aid readability, the content of the code is split across 5 files. These are 2 header files, ‘components.hpp’ and ‘circuit.hpp' and 3 .cpp files: ‘components.cpp’, ‘circuit.cpp’ and ‘main.cpp’. The .hpp files contain function declarations and class structures. The files ‘components.cpp’ and ‘circuit.cpp’ contain function definitions relating to their respective declarations in the .hpp files with the same namesake. The file ‘main.cpp’ creates a user interface into the terminal so the code can be utilised more easily.

2.1. Class structure

The code makes use of polymorphism with an abstract component base class (Figure 4). This contains: a virtual destructor, a function to calculate the impedance a component and functions to return the name of a component and its member data. This base class only has a string representing a component name in its member data. All other component classes (ideal and non-ideal) derive from this, adding member data relevant to each such as resistance and parasitic inductance. Non-ideal components derive from their ideal counterparts. The expressions used to calculate the impedance and phase shift of a component are given in Tables 1 and 2.

The ‘circuit’ class also derives from the component base class, as a circuit is made of components. This adds member data for the name of a circuit and a vector of containing pairs of components and their connection types. This class contains functions to calculate the impedance of the circuit. This class also overloads the insertion operator to print circuit details to the terminal. It can also create circuit diagrams.

The ‘two_circuit’ class derives from the circuit class as it is made of two or more circuits, or a circuit and a nested circuit joined together. This allows for the user to create nested circuits, or bigger nested circuits. This class contains private member data for the name of the nested circuit and a vector of the circuits it contains. The data members in the ‘circuit’ class were made private so that the ‘two_circuit’ class did not inherit them.

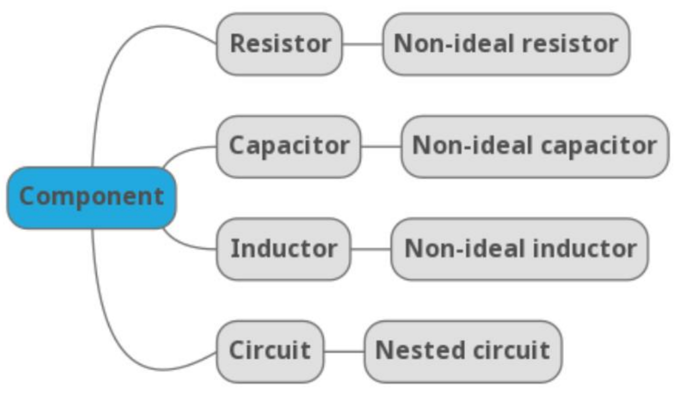


Figure 4 – Diagram of polymorphism made use of in the code. The component class in blue is the abstract base class.

2.2. User interface

The user interface consists of a while loop which contains a nested switch. The nested switch contains individual stack frames for each case so initialization of automatic objects is not bypassed. Outside of the while loop, vectors containing user created components, circuits and nested circuits are stored. This is so that created objects are not destroyed at the end of each stack frame or while loop iteration.

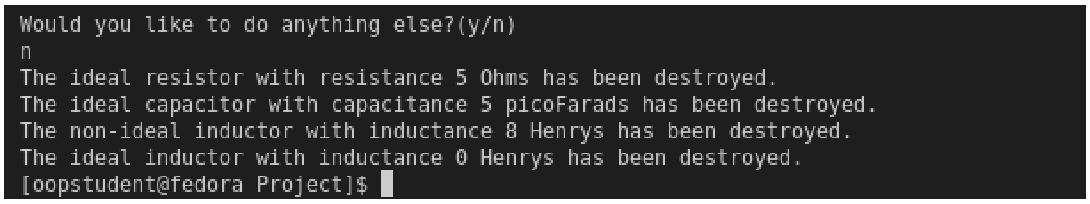
The switch statement allows users to choose whether they would like to:

1. Create a new component
2. Create a new circuit
3. Calculate circuit impedance
4. Create a nested circuit
5. Print a circuit
6. Exit the program

The first option in this list creates the necessity for a nested switch statement. After selecting option 1, users are asked to select the type of component they wish to construct. Their input is handled by another switch.

Error handling is used on user inputs to ensure correct data types are entered. Input is also validated so the user cannot access elements of the vector storing created components and circuits which are out of bounds. Inputs for components are also checked to be non-zero to avoid division by zero errors.

Components are created using *std::make_shared*. This allows for components to be wrapped in shared pointers, which is useful as components may be used in more than one circuit. Since there are no cyclical references between classes, smart pointers also avoid the risk of memory leaks (Figure 5).



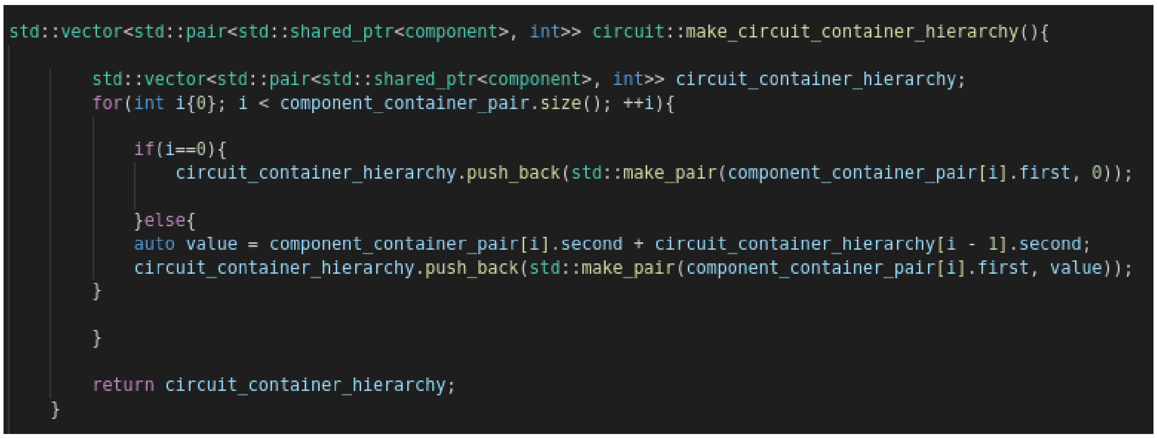
```
Would you like to do anything else?(y/n)
n
The ideal resistor with resistance 5 Ohms has been destroyed.
The ideal capacitor with capacitance 5 picoFarads has been destroyed.
The non-ideal inductor with inductance 8 Henrys has been destroyed.
The ideal inductor with inductance 0 Henrys has been destroyed.
[oopstudent@fedora Project]$ 
```

Figure 5 – The correct destructors called for the components created in the circuit used for impedance calculations in Figure 9.

The member data for the ‘circuit’ class containing pairs of components and their connection type is called ‘component_container_pair’. Due to the use of *std::make_shared*, it is a vector of shared pointers of type component, paired with an integer specifying the connection type. Users are asked to input 0 if they would like serial connections, or 1 if they would like connections in parallel. This is an important aspect for the impedance calculation.

2.3. Impedance algorithm

The impedance of a circuit is calculated in two parts. The first of these ranks components according to which should have its impedance calculated first. This is completed by ‘make_circuit_container_hierarchy’ (Figure 5). This function creates a running total of the sum of the second element in ‘component_container_pair’ (the connection types). It then outputs a vector with the same first elements (components) in each pair but with the second element as the running total at that point in the vector.



```cpp
std::vector<std::pair<std::shared_ptr<component>, int>> circuit::make_circuit_container_hierarchy(){

        std::vector<std::pair<std::shared_ptr<component>, int>> circuit_container_hierarchy;
        for(int i{0}; i < component_container_pair.size(); ++i){

            if(i==0){
                circuit_container_hierarchy.push_back(std::make_pair(component_container_pair[i].first, 0));

            }else{
            auto value = component_container_pair[i].second + circuit_container_hierarchy[i - 1].second;
            circuit_container_hierarchy.push_back(std::make_pair(component_container_pair[i].first, value));
        }

        }

        return circuit_container_hierarchy;
    }
```

Figure 5 – The ‘make_circuit_container_hierarchy’ function. The first component in a circuit always has a connection type of 0. This is the reason for the *if* statement in the function. The else function increments the running total.

The output of this function is then given to the ‘get_circuit_impedance’ function which calculates the impedance of the circuit recursively. This starts with the component possessing the highest value of the running total as its second pair. This component always lies at the end of the ‘make_circuit_container_hierarchy’ output. The function then evaluates the running total associated with the component ranked one less than the maximum. This can only be the same or less than the current component’s. If this running total is the same, then the impedance is added in series. Otherwise, it is added in parallel. This is then added to the circuit’s impedance and the function calls itself again, working through the components.

Nested circuits are created from other circuits connected in series. Therefore, the impedance of a nested circuit is calculated sequentially using the same function. Each constituent circuit's impedance is added to the last.

2.4. Circuit and nested circuit construction

Since users cannot go back to add a component in series with a component if they have already added a component in parallel with that component, the 'two_circuit' class was created. This allows for the creation of nested circuits by adding two circuits together in series (Figure 7). The class can create complex circuits as a circuit can be added to a nested circuit which has already been created.

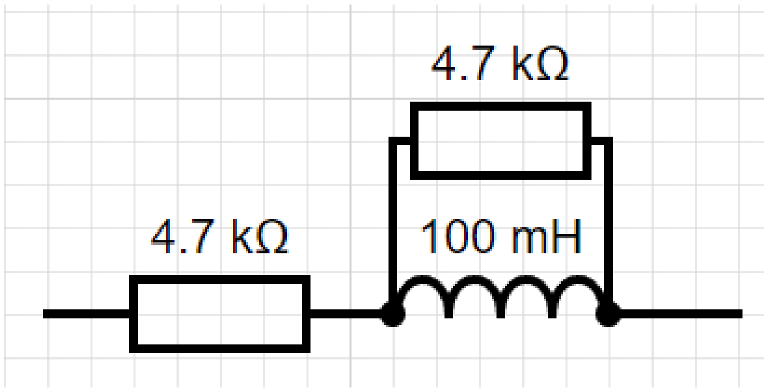


Figure 7(A) – A resistor in series with an inductor in parallel with another resistor.

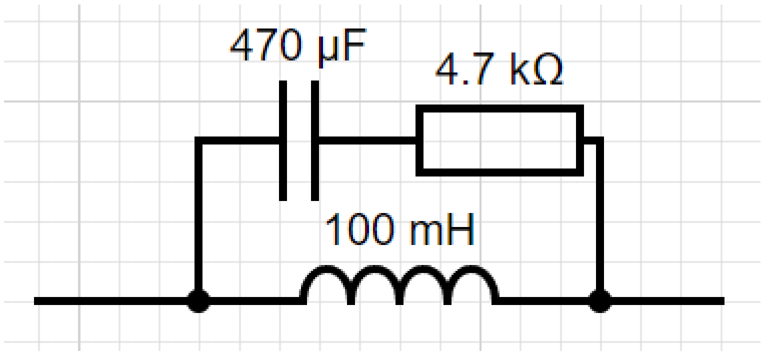


Figure 7(B) – A capacitor and resistor connected in series, nested in a parallel branch of an inductor.

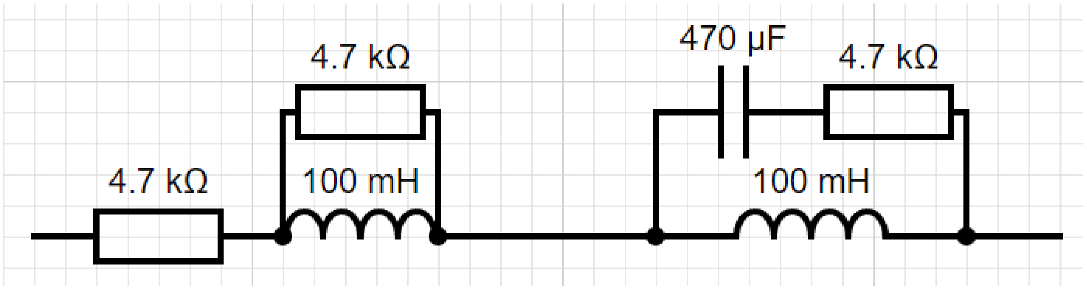


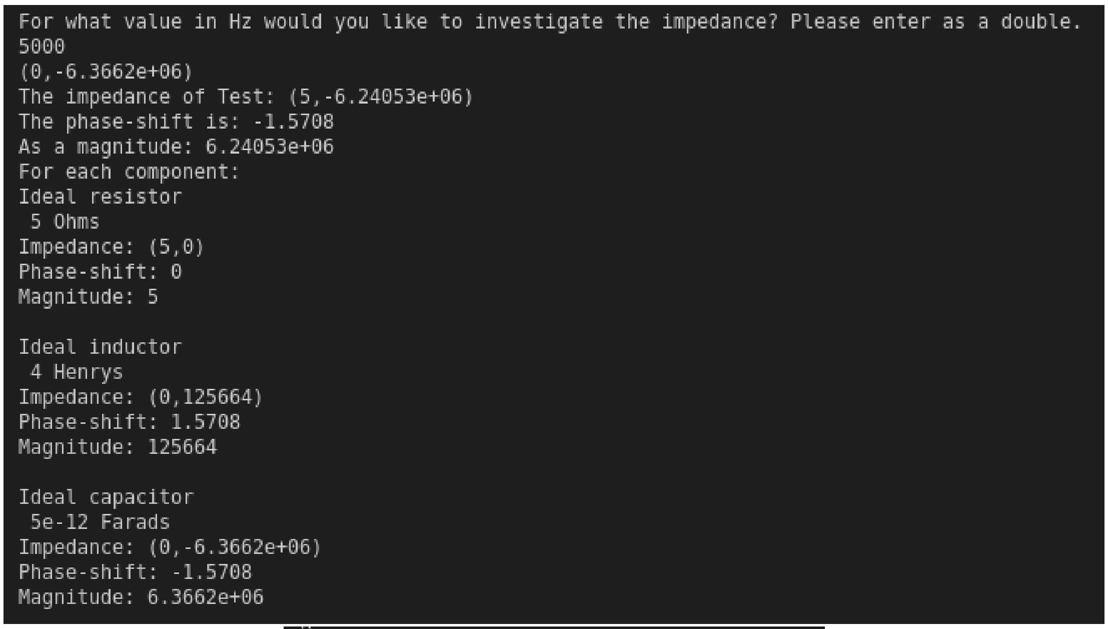
Figure 7(C) – A more complex circuit constructed from adding the circuits in Figures 7(A) and 7(B) in series.

The circuit in Figure 7(C) is still relatively simple and even more complex circuits can be constructed.

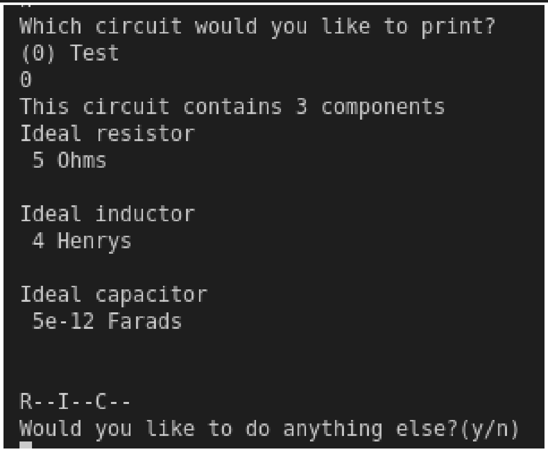
Inputs are validated to ensure the user cannot create nested circuits without first creating at least one circuit. Input is also checked to ensure a nested circuit is created before being able to add a circuit to a nested circuit.

3. Results

The code was first tested using a simple circuit comprising a resistor, inductor and capacitor (RLC) of 5Ω, $4H$ and $5pF$ connected in series. The circuit impedance was then calculated using a frequency of $5kHz$. The circuit details were also printed and a circuit diagram was produced.



```
For what value in Hz would you like to investigate the impedance? Please enter as a double.
5000
(0,-6.3662e+06)
The impedance of Test: (5,-6.24053e+06)
The phase-shift is: -1.5708
As a magnitude: 6.24053e+06
For each component:
Ideal resistor
 5 Ohms
Impedance: (5,0)
Phase-shift: 0
Magnitude: 5

Ideal inductor
 4 Henrys
Impedance: (0,125664)
Phase-shift: 1.5708
Magnitude: 125664

Ideal capacitor
 5e-12 Farads
Impedance: (0,-6.3662e+06)
Phase-shift: -1.5708
Magnitude: 6.3662e+06
```

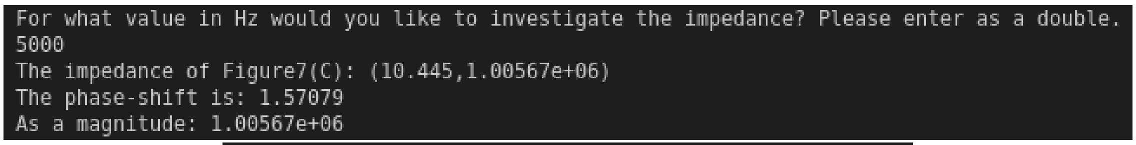


```
Which circuit would you like to print?
(0) Test
0
This circuit contains 3 components
Ideal resistor
 5 Ohms

Ideal inductor
 4 Henrys

Ideal capacitor
 5e-12 Farads


R--I--C--
Would you like to do anything else?(y/n)
```

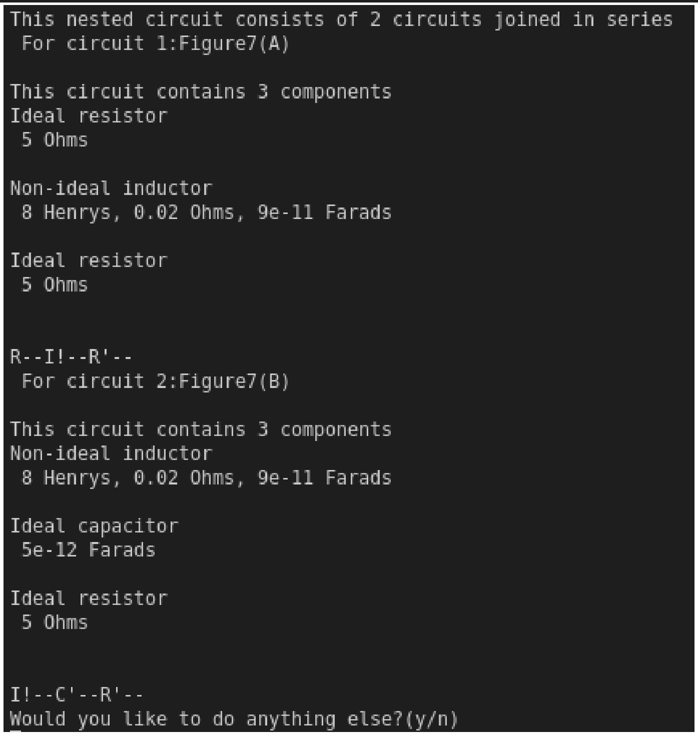
Figure 8 - The impedance result with the component details and circuit diagram for the RLC circuit. The result is calculated using the recursive impedance function. The component details are printed using the overloaded insertion operator. The circuit diagram is created using a separate function belonging to the circuit class.

The impedance calculation agrees with calculations carried out by hand using equation in Table 1 and from [4]. This therefore suggests that the impedance calculation is correct. Agreement was also found with other trialled circuits.

A more complex circuit was constructed after trialling the basic RLC case. This was constructed by joining the two circuits in Figures 7(A) and 7(B) to create the circuit in Figure 7(C). The details of its impedance calculation, circuit details and diagram are shown in Figure 9.



```
For what value in Hz would you like to investigate the impedance? Please enter as a double.
5000
The impedance of Figure7(C): (10.445,1.00567e+06)
The phase-shift is: 1.57079
As a magnitude: 1.00567e+06
```



```
This nested circuit consists of 2 circuits joined in series
 For circuit 1:Figure7(A)

This circuit contains 3 components
Ideal resistor
 5 Ohms

Non-ideal inductor
 8 Henrys, 0.02 Ohms, 9e-11 Farads

Ideal resistor
 5 Ohms


R--I!--R'--
 For circuit 2:Figure7(B)

This circuit contains 3 components
Non-ideal inductor
 8 Henrys, 0.02 Ohms, 9e-11 Farads

Ideal capacitor
 5e-12 Farads

Ideal resistor
 5 Ohms


I!--C'--R'--
Would you like to do anything else?(y/n)
```

Figure 9 – The impedance calculation for the circuit shown in Figure 7(C). This circuit is constructed with a non-ideal inductor to show the non-ideal component functionality. Non-ideal components are represented on the circuit diagram with an exclamation mark following their initial. The branch number a component resides on is shown by the number of apostrophes after their initial in the diagram. The diagram for a nested circuit is shown using separate diagrams for each constituent circuit to aid readability.

The impedance worked out by hand was found to be $10.246 + 0.9e - 6\, i\, \Omega$. The phase difference was found to be $1.57078\, rad$. Both of the program's results show strong agreement with the impedance calculations carried out by hand.

4. Conclusion

Polymorphism and inheritance were used in this code to calculate the impedance of an AC circuit. This relied on a class structure with an abstract component base class. The derived circuit class makes use of shared pointers to manage the components it contains without memory leaks. The insertion operator is also overloaded so that circuit details can be printed to the terminal. The user can also see a circuit diagram of the constituent components with their connection types if they would like. A user interface was also created so that the code could be applied easily by others.

A recursive function was used to calculate the impedance of a circuit. The results from this function show good agreement with the impedance calculated by other means. The code also allows for nested circuits to be created using a subclass of the circuit class.

To improve the functionality of the code, plots (using Gnuplot, for example) could be displayed to show the variation of impedance with frequency. This could be used to determine the resonant frequency of the circuit. A graphical user interface could also be created to allow for circuits to be visualised more easily. The component class could also be extended to consider current flow. This would extend the applicability of the code drastically as components such as diodes and transistors could be added.

References

[1] https://pwg.gsfc.nasa.gov/Electric/-E11-reason.htm accessed 25/05/2022

[2] Callegaro, L. (2012). *Electrical impedance: principles, measurement, and applications*. CRC Press.

[3] University of Michigan, National Science Foundation GOALI program, Electromagnetics Research Group, Non-Ideal Behaviour of Circuit Components

[4] https://www.translatorscafe.com/unit-converter/en-US/calculator/series-rlc-impedance/ accessed 27/05/2022